STYLE EMULATION OF DRUM PATTERNS BY MEANS OF EVOLUTIONARY METHODS AND STATISTICAL ANALYSIS

Gilberto Bernardes
Faculdade de Engenharia da Universidade do Porto
bernardes7@gmail.com

Caros Guedes
INESC, Porto
carlosguedes@mac.com

Bruce Pennycook
University of Texas, Austin
bpennycook@mail.utexas.edu

ABSTRACT

In this paper we present an application using an evolutionary algorithm for real-time generation of polyphonic drum loops in a particular style. The population of rhythms is derived from analysis of MIDI drum loops, which profile each style for subsequent automatic generation of rhythmic patterns that evolve over time through genetic algorithm operators and user input data.

1. INTRODUCTION

Application kin.genalgorthythm is a Pd patch that uses a genetic algorithm (GA) and statistical analysis data gathered from pre-existing MIDI drum loops in order to perform automatic variations on existing drumming styles. This is done in real time and is mainly intended for real-time performance driven by user-controlled data.

GAs are known for being a powerful technique for problem solving by searching in a vast space of possibilities, created from conventional genetic operators such as crossover and mutation, by means of a fitness function, which typically consists of an objective function that is able to rank all chromosomes in order to find an optimal solution. This method has been widely used as a creative tool in music, particularly for the development of variation of music sequences [1,2,3]. However, as noted by Biles [1], encoding a fitness function in a GA for real-time operation in music is a major issue due to the complexity of the aesthetic judgments related to this task.

Papadopoulos and Wiggins [5] present a categorization of musically-oriented GA applications based on the use of fitness functions. They establish a major difference between the use of an objective fitness function and a human one. In other words, they discuss the difference between evaluations of chromosomes based on formally stated computable functions, or in human judgment as a means of replacing the fitness function.

The solution we adopt here does not use a fitness function, and encodes several musical constraints directly in the GA's operators involved in the generation of new populations. This is done to avoid non-musical search spaces.

Our approach is inspired on Arne Eigenfeldt's *Kinetic Engine* [2,4] and focuses on two points: (1) an elaboration on the crossover technique proposed by Eigenfeldt; (2) the introduction of a metrical-supervision procedure on the mutation operator. Our elaboration on the crossover technique proposed by Eigenfeldt considers different lengths of the parental chromosome. The metrical supervision procedure operated on the mutation operator mutates the events according to the meter of the drum loop by using the metric indispensability principle presented by Clarence Barlow [6].

The output of the algorithm thus results from the selection of the best candidate from an evolving population of metrically coherent rhythms produced by the GA. Subsequently, the user can further control the musical output by introducing two parameters – density and complexity of events. These parameters can be introduced in real or non-real time, but they do not affect the metrical coherence or the style of the rhythmic sequence.

2. SYSTEM DESIGN

Below we present the design scheme for kin.genalgorhythm. The two blocks at the left (Analysis and Stored analysis) deal with previous analysis and storage of MIDI drum loops. The data gather and stored from these blocks will feed and initialize the main block (Generation of metrically supervised population) that uses a genetic algorithm containing two operators – crossover and mutation. This central block is responsible for the creation of a finite number of metrically coherent populations. Finally, at the right we have two blocks (User input and Performance), which deal directly with the performative aspects of the algorithm. These blocks have two roles. The first (User input) is to find the optimal solution from the generated offspring population, based on user input data – complexity and density. The second (Performance) appropriately distributes the rhythmic phrase among several drum parts.

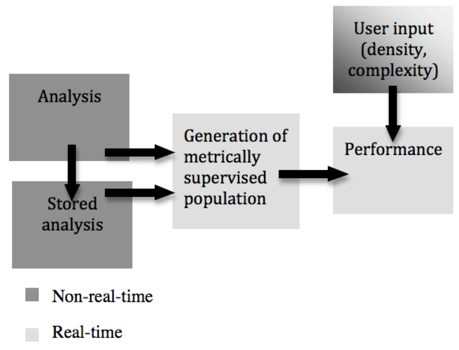
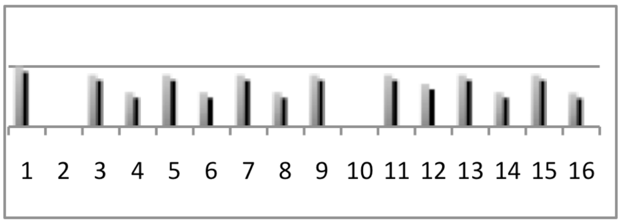

User input (density, complexity)
Analysis
Generation of metrically supervised population
Performance
Stored analysis
Non-real-time
Real-time

Figure 1. The design scheme for kin.genalgorithm.

3. INITIALIZATION

We do previous non-real-time analyses of polyphonic MIDI drum loops in a particular style in order to inform the algorithm about the specificities of that style. The analysis block of the algorithm will output the probability distribution of event occurrences for each individual part within a measure. In other words, each resulting vector corresponds to the normalized histogram of events that occur in each subdivision of a pre-defined measure.
A general vector, which takes into account all instrumental parts is also defined in order to gather the metrical accent distribution of the style under analysis. We use two different collections of MIDI loops containing style and tempo annotations available in the Free Pack from Groove Monkey [7] and Apple's Logic Pro [8]. In Figure 2 we can observe the normalized probability distribution for the Mambo loops from the Groove Monkey Pack [7].



1.00	0.00	0.86	0.0.57	0.86	0.57	0.86	0.57	0.86	0.00	0.86	0.71	0.86	0.57	0.86	0.57

Figure 2. Normalized probability distribution obtained with the analysis of the Mambo loops from the Groove Monkey Pack [7] of the 16 pulses comprising the 16th note level of a 4/4 meter.

Prior to the analysis, all the loops labeled with the same style (e.g. "70s Street Drums") are quantized, assembled together in a MIDI sequencer, and subsequently exported as a sole MIDI file. Our application uses a feature that can store the analyses and provide an instant access for future use.

The analysis block is dependent on three variables that have to be introduced by the user: (1) the tempo of the loop, indicated in beats per minute (BPM); (2) the length of the chromosome to be analyzed, which corresponds to the number of subdivisions in each measure (for example in a 4/4 meter at the 16th level, we would have 16 subdivisions); and (3) the amount of subdivisions each pulse contains (drawing on the previous example, and assuming that we have 16 subdivision, we would have to inform the algorithm that each pulse contains 4 subdivisions, in order to define a 4/4 meter at the 16th note level).

4. EVOLUTIONARY METHODS

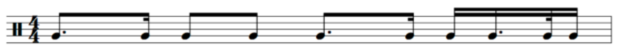
The evolution of the GA is mainly governed by two principles: crossover and mutation. In this section we present the transformations we operated in principles advanced by Eigenfeldt [2,4] as well as a novel model to metrically supervise the mutation operator.

4.1 Crossover

Crossover is a standard evolutionary technique in GA, in which portions of two individuals (parents) are spliced together at random split points and rejoined interchangeably in order produce a new variation (children). When combining the two parental chromosomes the idea behind this operation is that the resulting chromosomes may be better than both of the parents if it takes the best characteristics from each of the parents.

As Eigenfeldt points [2,4], crossover is not a usual developmental technique in music due to the arbitrary method used to determine the splitting point. In order to circumvent this problem, Eigenfeldt proposes the use of a single parent chromosome in which a first-order Markov chain is applied to perform the crossover. However, since most styles analyzed here do not present much variation, we found a tendency to obtain almost the exact same sequence in all offspring populations. Several experiments showed us that if we increase the dimension of the parental chromosome, the amount of novelty increases as well. Therefore, an average length 60 beats (around 30 seconds of a sequence at 120 BPM) revealed much more proficient results.

The resulting chain of events is based on a transition table, which gives the probability of moving from one state to another. The pairs of successive rhythmic cells are coupled using the object **anal** that computes the transition matrix. Object **prob** is then used to generate the events according to the transition matrix. Both objects belong to the cyclone external library in Pd-extended [9]. In order to compute the transition matrix, we convert the binary representation of each rhythmic cell of the offspring chromosomes sequences to decimal (see Figure 3). When making the analysis, the user already defines the length of each rhythmic cell. Our method ignores the duration of each note, as this does not affect the outcome – we are dealing with percussive instruments with a natural decay. Special attention is paid to beginnings of offspring phrases, which are restricted to the first rhythmic cell of the analyzed material.



10000010	10001000	10000010	10100110
138	104	138	142

Figure 3. Example of an offspring in musical, binary, and decimal representations.

4.2 Mutation

The purpose of mutation is to introduce and preserve diversity in the offspring population in order to access a wider range of possible musical densities. A common usage of this technique, involves altering a certain number of arbitrary bits in a genetic sequence depending on the level of desired mutation.

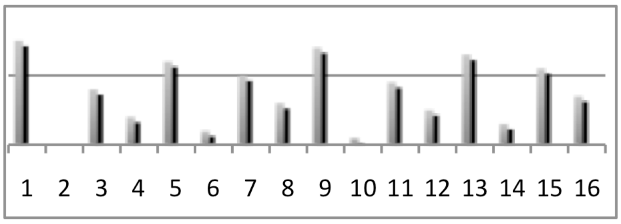
The method we use is a variation of a stochastic algorithm, usually known as roulette-wheel selection (RWS). For each bin, a random number is generated and compared against a segment division, deciding if the subdivision in cause should be mutated or not.

In our approach, we bias the mutation operator in order to produce a metrically coherent output. This is done using a decision-making process that takes into account Barlow's metrical indispensability principle [6]. The amount of mutation is assigned by default to the level of desired density as input by the user. Low and high densities are assigned to low and high mutation ratios respectively. The metrical coherence of the offsprings is preserved independently from the mutation ratio.

4.2.1 Metrically-supervised Mutation

Barlow's metric indispensability principle defines the probabilistic weight each accent at metrical level on a given meter should have in order for that meter to be perceived – i.e. how *indispensible* is each accent at a certain metrical level for a meter to be felt.

The accents' weights are calculated by a formula that takes into account the meter (e.g. 4/4) and the metrical level (e.g. 16th note) for which one wants to calculate the indispensabilities. The metrical level is defined by a unique product of prime factors which equals the number of pulses at that metrical level, and takes into account the division (binary or ternary) at higher levels. For example, the six pulses comprising the 8th-note level in a ¾ meter would be defined as 3x2 (representing the *three* quarter notes at the quarter-note level that subdivide into *two* 8th-notes at the level below), whereas the six pulses comprising the 8th-note level in 6/8 would be represented as 2x3 (*two* dotted quarters that subdivide into *three* 8th-notes). Below we show the normalized distribution for the 16 pulses comprising the 16th note level in 4/4.



1.00	0.00	0.53	0.27	0.80	0.13	0.67	0.40	0.93	0.07	0.60	0.30	0.87	0.02	0.73	0.47

Figure 4. Probability distribution given by Clarence Barlow's indispensability formula for the 16 pulses comprising the 16th note level of 4/4, which is defined as 2x2x2x2.

Barlow's metrical indispensability algorithm has been effectively used to automatically generate rhythm at a certain meter. In this application, we use these distributions as a metrical template that is applied to the mutation operator of the GA, by supplying the threshold mutation values for each of the measure subdivisions.

Aside from metrical indispensability, we implemented two other different vectors that can be used instead. One is the probability distribution vector of each analyzed style, and the other is the mean of the product of the last vector and the metric indispensability vector. Further research should explore other possible metrical templates by using other vectors. Empirically, we observe almost the same quality results with the current three possibilities.

5. PERFORMANCE

Initially, the user must define the style she or he wants to perform; either by selecting one of the pre-stored analyses or by analyzing a midi drums sequence. In order to create variation at the macro structural level the user can assign values for two parameters (density and complexity), either in real- or non-real time.

5.1 Density and complexity parameters

Whenever a genetic operator (crossover or mutation) is applied to create offspring populations, the values of density and complexity for each offspring are calculated.

The density of each rhythmic phrase corresponds to the rate of the attacks per measure. For example, if we have the following string: [1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1], the density is calculated by diving the sum of all the string values (10), and the total number of values of the string (16), which gives a density of 0.625.

The complexity parameter specifies the degree of syncopation by weighting the events occurring at the weaker pulses of the metric structure. It is calculated in three steps: 1- by multiplying each offspring vector by the symmetric of the indispensability vector around 0.5 (i.e. by subtracting each value from 1.0); 2- by summing all the values resulting from the operation; and 3- by dividing the sum by the number of elements in the vector. This way, vectors containing more events on weaker pulses of the metrical level (i.e. more syncopated) have higher complexity values. Figure 5 describes the procedure.

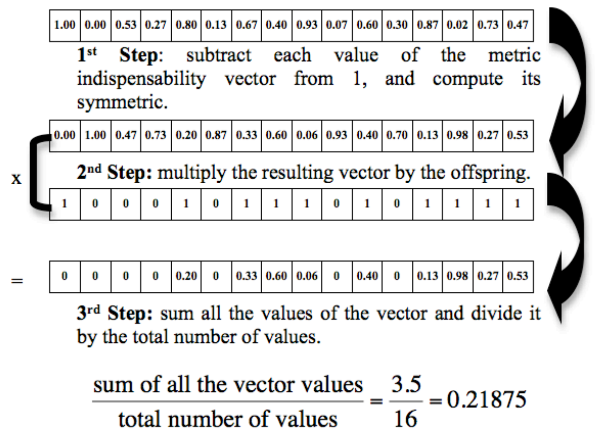

1.00 0.00 0.53 0.27 0.80 0.13 0.67 0.40 0.93 0.07 0.60 0.30 0.87 0.02 0.73 0.47
1st Step: subtract each value of the metric indispensability vector from 1, and compute its symmetric.
0.00 1.00 0.47 0.73 0.20 0.87 0.33 0.60 0.06 0.93 0.40 0.70 0.13 0.98 0.27 0.53
x
2nd Step: multiply the resulting vector by the offspring.
1 0 0 0 1 0 1 1 1 0 1 0 1 1 1 1
=
0 0 0 0 0.20 0 0.33 0.60 0.06 0 0.40 0 0.13 0.98 0.27 0.53
3rd Step: sum all the values of the vector and divide it by the total number of values.
sum of all the vector values / total number of values = 3.5 / 16 = 0.21875

Figure 5. Steps for computing the complexity of each vector.

For each offspring population the program computes a table like the one shown below.

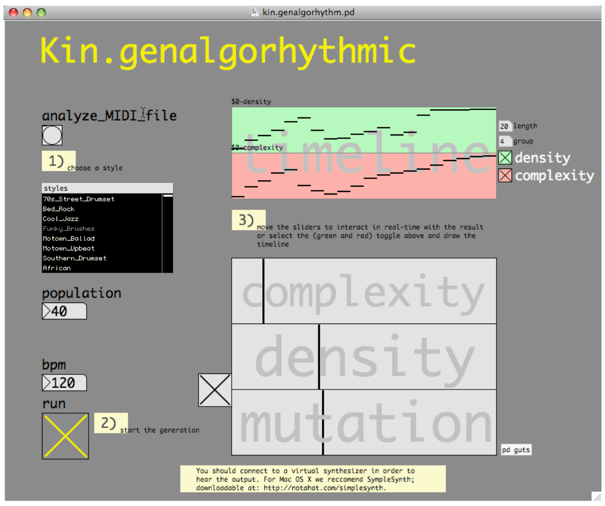
Sequence	Density	Complexity
10000010 10001000 10000010 10100110	0.3125	3.77419
10000010 10101110 10001010 10001010	0.40625	4.93548
10000000 10001000 10000000 10100010	0.21875	3

Table 1. Example of a density and complexity measurements.

5.2 Selection for Performance

The user must provide values corresponding to the density and complexity of the phrase for performance, so that the algorithm at runtime chooses offsprings that are closely related to the values input by the user.

The selection is done by finding the offspring that presents the highest correlation between a vector defined by the two values given by the user (density, complexity), and the correspondent vectors for all generated population. The density and complexity values can be assigned and stored previously in a table or altered in real-time. In Figure 6 we can observe the two different input approaches: the non-real time methods that can be stored in a timeline (graph on the top right of the window), and the real-time method by directly adapting the density and complexity sliders (below the timeline). The algorithm reads a new value on the beginning of each new sequence.



3) move the sliders to interact in real-time with the result or select the (green and red) toggle above and draw the timeline

You should connect to a virtual synthesizer in order to hear the output. For Mac OS X we reccomend SympleSynth; downloadable at: http://notahat.com/simplesynth.

Figure 6. Main window in Pure Data presetting the interface where the user can both store the complexity and density values in a timeline (in the upper right) or alter the values in real-time directly on the sliders (below the timeline).

5.3 Performance

At runtime, the algorithm (1) will sequentially read the selected offspring, and (2) will evaluate in all probabilistic distribution tables (each corresponding to a different part: i.e. snare drum, hi-hat) to select if that part will play or not.

We felt the necessity to implement two rules that constrain the output in order to avoid certain non-musical situations. The first was to avoid the coincidence of similar instruments, and the second was to avoid the excessive appearance of phrase beginning accents such as crash-cymbal hits. In order to solve the first situation we restricted similar instruments to play in the same subdivision of the beat, opting for the strongest/sharpest one – e.g. if we have for the same beat a snare drum and a steel snare drum, we will only play the second one. For the second situation, we restricted the appearance of strong accents (such as crash cymbals) to the first of each four phrases.

A special attention should be paid to the tempo of the generative patterns, which by default is assigned to the tempo the user has introduced during the analysis. Certain styles have a clear BPM range associated with it, and this factor can determinant for the perception of a certain style. For example if we consider styles such as House music and Funk, the tempo difference is crucial to denote the difference between them, since they have almost the same generative profile.

6. CONCLUSION

In this paper we present an application that generates rhythmic patterns in a specific style by means of a genetic algorithm and statistical analysis. Major differences from similar software include the focus on the emulation of different styles and special attention is given to the metrical coherence of the output. The use of Barlow's indispensability algorithm [6] proved to be an efficient method for assuring metrical coherence in the offspring generation by mutation.

The software, analyses and generated samples used in the study are available at: http://sites.google.com/site/kineticproject09/home.

7. ACKNOWLEDGMENTS

This joint research project is made possible thanks to the support from the Portuguese Foundation for Science and Technology (grant UTAustin/CD/0052/2008) and the UT Austin | Portugal Program in Digital Media [10].

8. REFERENCES

[1] Biles, J. (1994). GenJam: A Genetic Algorithm for Generating Jazz Solos. *Proceedings of the International Computer Music Conference*

[2] Eigenfeldt, A. (2009). The Evolution of Evolutionary Software Intelligent Rhythm Generation in Kinetic Engine. *Proceedings of EvoMusArt 09, the European Conference on Evolutionary Computing*, Tübingen, Germany.

[3] Martins, J. and Miranda, E. (2007). Emergent rhythmic phrases in an A-Life environment. *Proceedings of ECAL workshop on Music and Artificial Life*, Lisbon, 2007.

[4] Eigenfeldt, A. (2006). Kinetic Engine: Toward an Intelligent Improvising Instrument. *Proceedings of Sound And Music Computing*, Marseille, France.

[5] Papadopoulos, G. and Wiggins, G. (1999). AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects," *Proceedings of the AISB'99 Symposium on Musical Creativity*, Edinburgh, UK.

[6] Barlow, C. (1987). Two essays on theory. *Computer Music Journal*, 11, 44-60.

[7] http://www.groovemonkee.com/home/

[8] http://www.apple.com/logicstudio/

[9] http://puredata.info/

[10] http://www.utaustinportugal.org/